Analysis of distributed multi-periodic systems to achieve consistent data matching

Nadège Pontisso, Philippe Quéinnec, Gérard Padiou

SPECIAL ISSUE PAPER

Analysis of distributed multiperiodic systems to achieve consistent data matching

Nadège Pontisso, Philippe Quéinnec*,† and Gérard Padiou

Institut de Recherche en Informatique de Toulouse, Université de Toulouse ENSEEIHT, 2 rue Camichel, BP 7122, F-31071 Toulouse Cedex 7, France

SUMMARY

The distributed real-time architecture of an embedded system is often described as a set of communicating components. Such a system is dataflow (for its description) and time triggered (for its execution). The architecture forms a graph of communicating components, where more than one path can link two components. Because the characteristics of the network and the behavior of intermediate components may vary or are only partially known, these paths often have different timing characteristics, and the flows of information that transit on these paths reach their destination at independent times. However, an application that seeks consistent values will require these flows to be temporally matched so that a component uses inputs that all (directly or indirectly) depend on the same computation step of another component. In this paper, we define this temporal data-matching property, both in a strict sense and in a relaxed way allowing approximately consistent values. Then, we show how to analyze a system architecture to detect situations that result in data-matching inconsistencies. In the context of multiperiodic systems, where components do not necessarily share a common period, we also describe an approach to manage data matching that uses queues to delay too fast paths and timestamps to recognize consistent data sets.

KEY WORDS: distributed system; component-based architecture; real-time; data consistency

1. INTRODUCTION

Distributed systems are often built by assembling components that are independently developed or off-the-shelf, and the designer is faced with various challenges, especially when real-time is involved [1]. Techniques have been proposed to solve interconnection difficulties [2], such as wrapping to expose a regular interface. In a real-time context, these components must be appropriately scheduled using periods, deadlines, priorities, and so on [3]. Nevertheless, some problems remain when multiple paths connect two components. Indeed, the correct behavior of a component depends on correct or valid inputs. Independently of the semantic constraints of the inputs (e.g., belonging to a specific range of values), the time validity is also an important parameter in embedded systems. This time validity is often described in terms of availability (having inputs at the right time to start a task) and freshness (having recent enough inputs). Some works have studied the case where a component uses several inputs, and these inputs respect a time consistency constraint such as having been produced at the same time. But, this constraint is not sufficient: In a complex architecture, an intricate component graph leads to several paths between two components. In such a case, inputs of a component depend on the outputs of the *same* component (a source) by *several paths*. As a

*Correspondence to: Philippe Quéinnec, Institut de Recherche en Informatique de Toulouse, Université de Toulouse ENSEEIHT, 2 rue Camichel, BP 7122, F-31071 Toulouse Cedex 7, France.

†E-mail: philippe.queinnec@enseeiht.fr

path links several components that consume, transform, and produce data, this dependency is not on the source value itself but on the step at which it was produced. Our work fits in the following problem: How can the inputs of a component be consistent with regard to the production step of another component in the situation where several independent paths link these two components?

The data consistency is achieved by delaying fast paths until an adequate matching of inputs is possible. We approach this problem by analyzing the component graph to identify structures where two components are linked by several paths. If two paths have a really asymmetric nature, buffers are used to introduce a delay on the fastest path. In the general case, queues are used to keep data until the slowest data have arrived. As all values are not necessarily useful, we introduce filtering queues that keep only part of their inputs. We present results on the size of the required queues. These results are obtained in the context of periodic components but make neither assumptions nor constraints on the scheduling. The system is actually multiperiodic as components do not necessarily share a common period.

The paper is organized as follows. Section 2 presents related work. Section 3 introduces an extensive example, describes what is a consistent data matching, and defines the computation and communication model. Section 4 presents the analysis of the component graph. Section 5 describes data consistency management, the queue size computations, and the application on the example. Finally, Section 6 presents concluding remarks and outlines future directions.

2. RELATED WORK

In a real-time system, the freshness of data is a standard property. Freshness means that the system uses values that are as recent as possible or in a specific domain of time validity. But, this freshness property is not enough for some applications. Let us consider a toy example (Figure 1). This system computes $2x + 3(x + 1)$, where x comes from an initial component C_1; the output of C_2 is its input plus one; C_3 and C_4 multiply their input by a constant, and a last component C_5 adds the results of theses multiplications. When C_1 emits a flow of values, C_5 must not carelessly mix values coming from C_3 and C_4 but has to add values corresponding to the same x. If it behaves like this, we say that C_5 does a consistent data matching of its inputs. If the lower branch computation takes longer than the upper branch (e.g., twice as much time), using freshness only leads to inconsistent results as the most recent values that reach C_5 are not related to the same value of x.

Such a system fits well in the synchronous dataflow (SDF) paradigm [4, 5]. SDF is a special case of dataflow, where a program is represented by a directed graph in which each node (called block) represents a computation and each edge specifies a First In, First Out buffer. In the SDF paradigm, the execution of a block is enacted when it has enough inputs. The objective of the static analysis of an SDF program is to find the necessary buffers between blocks and a scheduling such that a block is executed when its inputs are available.

Data matching is not the goal of SDF graphs. In this toy example, as the system is a pure dataflow, consistent data matching can be obtained using SDF analysis. However, SDF theories cannot be used if the system is not a pure dataflow system, which is to say if the components are fired on the basis of conditions other than token availability. Especially, SDF cannot be used if the components are time triggered.

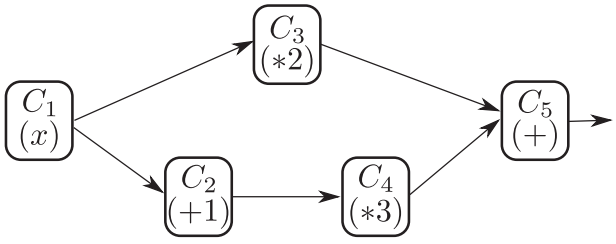

Figure 1. Computation of $2x + 3(x + 1)$.

Moreover, forcing a scheduling to solve this data-matching problem may be incompatible with other constraints, such as resources consumption or CPU availability, that are traditionally solved by scheduling analysis: Our goal is to analyze a system without considering a scheduling or a specific scheduler, neither do we want to compute a scheduling.

In the field of dataflow or database, studies mainly focus on the freshness of data (for instance [6], [7], or [8] for an extensive list of references). In [9], the authors determined an algorithm that computes which data need to be up-to-date taking data relationships into consideration. In [10], the variable semantics and their timed validity domain are used to optimize the transaction scheduling in databases. In [11], Object Constraint Language constraints are used to define the validity domain of variables, and a variation of Timed Computation Tree Logic is used to check the system behavior and to prevent a value from being used out of its validity domain. However, these works did not consider consistency of *sets* of values.

In [8], the authors introduced a 'mutual consistency' between objects in a database. They recognized that guaranteeing individual freshness of objects is insufficient as objects may be related to one another and that the system should present a logically consistent view of the objects. Let us imagine a system that independently reads sensors; at any time, this system may have fresh values for every sensor, although these values were acquired at different dates, and the set of values itself does not correctly reflect the actual reality, be it now or in the past. The paper deals with nonpreemptible periodic transactions, and the authors seek either the right periods and relative deadlines that would guarantee mutual consistency or if a given set of transactions with their known parameters guarantee mutual consistency. In a sense, they are looking for a correct scheduling of actions so that mutual consistency is preserved. Our work differs from theirs in that we make similar assumptions concerning the scheduling but have no influence on it.

In [12], the authors did a similar work distinguishing *image objects* and *derived objects*. Image objects are periodically sampled from outside sensors, and derived objects are computed from the values of a set of objects. The *age* of an image object is directly linked to its last reading, and the *age* of a derived object does not come from the date at which the computation occurs but is equal to the age of the oldest object that is used to compute it. To capture a mutual consistency constraint on the set of values used to compute a derived object, the authors introduced the notion of *dispersion*, which is the maximal difference between the ages of any two objects in a set. Then, a set of objects is *absolutely temporally consistent* if the age of all the objects is below a given absolute threshold; this set is *relatively temporally consistent* if its dispersion is below a given relative threshold. Given a set of periodic preemptible transactions that read image or derived objects and update derived objects, the authors' goal is to find which concurrency control strategy (among pessimistic, aka two-phase locking, and optimistic) and which scheduling (among rate-monotonic and earliest-deadline-first) perform the best. Again, the goal is to find a correct and efficient scheduling of the transactions.

This mutual consistency is also studied in the web domain [13]. The goal is to guarantee the weak consistency of replicas (cached values on a proxy web) with respect to the original data (pages of a web server). The authors observed that all the components that form a page (html text, images, and style sheets) must be consistent so that the user sees the current version or a past version but not a mix of different versions. They proposed an algorithm to find an adequate rate of polling of the server to individually invalidate cached replicas so that they do not diverge too much. Their algorithm is then adapted to maintain the weak consistency of a set of elements of pages. The quality of the consistency is measured by a notion similar to the dispersion, as seen earlier. The simplicity of the architecture (one proxy) and of the problem (only one version of each object, without history) leads to a simple and efficient algorithm.

Consistency in distributed systems is also an old problem. However, it is mainly performed from a logical point of view, yielding causal or total order of operations to ensure consistency of values. Some works exist that introduce real-time constraints in broadcasting. For instance Δ-causal protocols ensure the causal consistency of messages arriving by Δ. Research on this topic [14] has concentrated on adaptation issues (adjusting Δ) and optimizing the transmission (reducing the bandwidth overhead by minimizing piggybacking information). The goal of Δ-causality is to favor latency even if ignoring a too late message leads to breaking causality chains. In our case, we seek

a consistent matching of messages traveling by different paths. Latency is imposed by the slowest path, and messages on faster paths are delayed to enable this matching.

Our work differs from the results presented earlier mainly because our goal is not to compute a system scheduling to solve our problem of data matching. Neither do we consider that we know the final scheduling of components nor their implantation (for example, the number of CPU). This approach allows to manage systems composed by black boxes that we cannot constrain to have a 'good' behavior. For example, we cannot impose when the components read their inputs. Moreover, even with a configurable system, acting on scheduling can be insufficient to solve data-matching problems.

Prior work was carried out considering same frequency components [15], and it used solutions similar to SDF. In this paper, we consider multiple frequency systems, and it brings forth radically different solutions. An outline of the general analysis was presented in [16, 17].

3. CONSISTENT DATA MATCHING

3.1. Application example

Our application example comes from the FUEGO project. The component graph was developed in collaboration with Thales Alenia Space. FUEGO objective is to detect fires and eruptions and to observe their evolutions. The system has been conceived as a constellation of satellites in low earth orbit. Each satellite is equipped with an observation instrument (a narrow area sensor) and with a detection instrument (a wide area sensor) that is pointed in front of the satellite. The detection instrument detects fires or eruptions. In such a case, an alarm is sent to a ground mission center, and the satellite is requested to do an observation of the zone as soon as possible. A ground control center gathers all observation requests and allocates them between the satellites of the constellation.

We only study the system part in relation with the wide area detection instrument (Figure 2). The Global Positioning System allows to compute the satellite position. The wide area sensor takes pictures that are linked to compose an image of a wide area. The data sent by the gyroscope and the star tracker are used to compute the satellite attitude (the angle the satellite makes with the earth).

The hot point coordinate computation is made using the image, the satellite position, and its attitude. Having detected a hot point, the system computes its amplitude and its nature (fire or eruption). The amplitude and the coordinates of the hot point and the position that the satellite had when this point was detected are sent to the ground by the alert management component. The hot point management component analyzes the coordinates of the hot point, its nature, and its amplitude. It records the point if it is new or if it has evolved.

Using the hot point parameters and the actual parameters of the satellite, a component computes the date when the observation instrument flies over the hot point. On this date, the point nature and its coordinates are stored by the request management component. It schedules the hot point observations the satellite has to achieve. Table I displays the parameters of our system. In the numeric applications in this paper, we use null communication times between components to simplify the presentation. The actual analysis uses nonnull values, but the results are not significantly different.

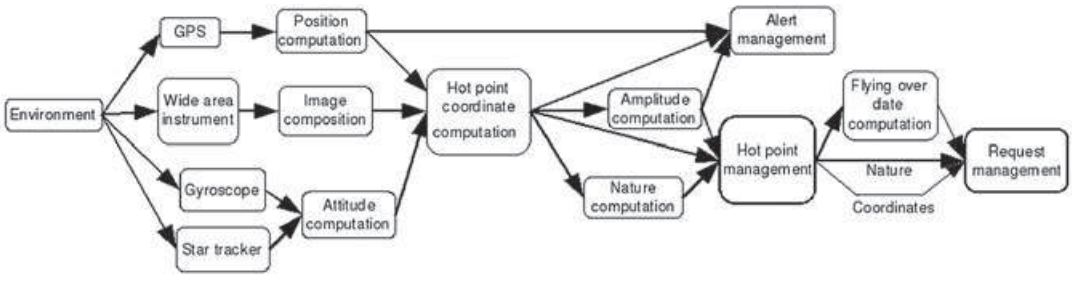

Figure 2. Application: a fire detection satellite. GPS, Global Positioning System.

Table I. Application example parameters.

Component	Period (in ms)	Minimal execution time (ms)	Maximal execution time (ms)
GPS	1000	100	200
Position computation	60	20	40
Alert management	1000	50	200
Wide area instrument	100	60	70
Image composition	1000	200	400
Coordinate computation	1000	100	500
Amplitude computation	1000	20	30
Nature computation	1000	30	40
Hot point management	1000	50	200
Flying over date computation	1000	30	30
Request management	1000	50	150
Gyroscope	60	20	30
Star tracker	120	40	60
Attitude computation	60	20	30

GPS, Global Positioning System.

3.2. Examples of data-matching inconsistencies

In Figure 2, the alert management has to send to the ground a message composed of three values: the coordinates of the detected hot point, its amplitude, and the position that the satellite had when this point was detected. The coordinate computation needs the satellite position to produce the coordinates. The amplitude computation needs the coordinates; hence, it also indirectly depends on the satellite position. Thus, the alert management uses three values that depend on the position produced by the position computation component. This set of three values is considered as consistent if the values depend on a same computation step of the position computation.

Similarly, the hot point management uses the coordinates, the amplitude, and the nature (fire or eruption) to find if it is a new hot point or an already detected one. In the later case, the component also determines the hot point evolution. It is important that all three inputs correspond to the same instant of capture of the same hot point. For instance, if two nearby hot points occur, the hot point management must not receive the coordinates of one of them and the amplitude of the other one. The parameters in Table I show that there is enough spreading in execution times to induce such a case, especially when we do not want to assume anything about the actual scheduling.

3.3. Consistency formalization

We consider a distributed computation that is modeled by sending events (noted s), delivery events (noted d), and internal events (noted i). We note s_C, d_C, or i_C as an event occurring on a component C. We note $d^{C'}$ as a delivery event corresponding to the reception of a message coming from the component C' and $d_C^{C'}$ as a delivery event occurring on C and corresponding to a message coming from C'. The internal events correspond to computation steps, and we consider that their durations are (logically) null. We note $\prec$ as the relation of temporal precedence between events on a given component.

3.3.1. Direct influence relation. The direct influence relation $\rightarrow$ is defined by the following:

- For a message m, the sending event influences its delivery event:

$$s(m) \rightarrow d(m).$$

- An internal event influences the sending events that directly follow until the next internal event:

$$\forall s_C, i_C : i_C \prec s_C \wedge \nexists i'_C : i_C \prec i'_C \prec s_C \Rightarrow i_C \rightarrow s_C.$$

- The last delivery event coming from a given component influences the following internal events until the next delivery event coming from the same component:

$$\forall d_C^{C'}, i_C : d_C^{C'} \prec i_C \wedge \nexists D_C^{C'} : d_C^{C'} \prec D_C^{C'} \prec i_C \Rightarrow d_C^{C'} \rightarrow i_C .$$

3.3.2. Influence relation. The influence relation, noted $\rightarrow^*$, is constructed by transitive closure of $\rightarrow$.

This influence relation is stronger than the usual causality relation (also called happened-before relation): If a influences b, then a causally precedes b; the converse is not necessarily true. The influence relation is closer to a memory model description of a distributed system than to a message-passing one. Note that $i \rightarrow^* i'$ if and only if there exists a sequence of the form $i \rightarrow s_1 \rightarrow d_2 \rightarrow i_2 \rightarrow s_2 \rightarrow d_3 \rightarrow i_3 \cdots \rightarrow i'$.

3.3.3. Influence past. We define the influence past of an event i as the set of internal events that influence i added to itself:

$$\text{past}(i) \triangleq \{i' \mid i' \rightarrow^* i\} \cup \{i\}.$$

3.3.4. Strictly consistent execution. We note $S \mid C$ as the subset of events from the set S that occur on component C. An internal event set is consistent if it contains at most one internal event by component. An execution is strictly consistent if the influence past of each internal event is a consistent event set:

$$\text{Strictly consistent execution} \triangleq \forall i : \forall C : \text{cardinality}\,(\text{past}(i) \mid C) \leqslant 1. \tag{1}$$

3.3.5. Relaxed consistency. We consider that each component has a real-time clock. We note $\text{date}(i)$ as the time at which the internal event i occurs. We call $\text{span}(S)$ the maximum time span between events in S:

$$\text{span}(S) \triangleq \max_{i_1, i_2 \in S} (\text{date}(i_1) - \text{date}(i_2)).$$

Then, we define a τ-relaxed consistent execution by the following:

$$\tau\text{-relaxed consistent execution} \triangleq \forall i : \forall C : \text{span}(\text{past}(i) \mid C) \leqslant \tau. \tag{2}$$

A 0-relaxed consistent execution is actually a strictly consistent execution. Note that in this definition, we compare dates of events that are all on the same component: A global synchronous clock is never required.

3.3.6. Consistent data matching. If we consider data instead of events, we say that a value d produced by an execution step S influences a value d' produced by a step S' if the internal event corresponding to S influences the one corresponding to S'. A data set is consistent if the union of the influence pasts of the internal events that produce the data is consistent. A component does a consistent data matching if its inputs form a consistent data set for each execution step.

3.4. Model

We solve our data-matching problem in a general setting that does not depend on an effective scheduling or a particular scheduler. We define a general abstract model that grabs just enough requirements to solve our problem without restricting too much the systems where the solution is applied.

3.4.1. Computation model. Components are time triggered, and we impose that they have a fixed period. Different components may have different periods. During one step of its period, the component reads exactly once every input port; then it performs its computation, and then it writes exactly once every output port. The only requirement is that a component finishes its step before the end of its period. These weak assumptions allow to fully abstract any scheduling considerations. A component step can be instantaneous or can take as long as the full period. Preemption may split it into pieces. In consecutive periods, component steps may have different durations or different relative start times (variable phases). Different readings of one step can be performed instantaneously or separately and similarly for writings.

3.4.2. Communication model. In the same spirit, we make few assumptions about communication. We assume that communication is First In, First Out and reliable. We use a minimum and a maximum communication time. These boundaries are defined for each couple of components and can vary in the system. By allowing null values, we model a nontransactional memory. On the other hand, nonnull values model a communication network. The strict upper bound is natural in a real-time context, for instance, when communication is performed via a synchronous bus.

3.4.3. Model parameters. To analyze the queue sizes of a system, some parameters are useful. The mandatory parameters are the following:

- T_C : the period of component C.
- $\Delta_{CC'}$: the maximum communication delay between components C and C'. An upper bound is sufficient.

Optional parameters are the following (may be set as 0 if unknown):

- e_C : a lower bound of the execution time of a step of component C.
- $\delta_{CC'}$: the minimum communication delay between components C and C'. A lower bound is sufficient.

4. SYSTEM ANALYSIS

4.1. Graph analysis

To analyze the system, we analyze the component graph as an oriented graph. We are able to easily found problematic configurations searching for subgraphs that we called spindles, which detect that several paths exist between two components.

4.1.1. Graph properties

Simple path. A path is a sequence of nodes where there exists an edge between two consecutive nodes. A simple path is a path in which all nodes are distinct.

Separated paths. Two simple paths with the same extremities are separated if and only if their sequences do not have any nodes in common, except for the initial and final nodes.

4.1.2. Spindle. A *spindle* between two nodes is the set of all simple paths connecting these nodes such that at least two separated paths exist in this set. The initial node of these paths is called the *source*, and their final node is called the *sink*. In Figure 2, the set of the three simple paths between the position computation component and the alert management is a spindle.

Theorem 1
An inconsistent data matching can occur between a component couple (C, C') if and only if there is a spindle between them.

Proof
If a spindle exists between the two components C (the source) and C' (the sink), two (or more) separated paths exist. The data that were influenced by C reach C' by using different noncoordinated paths, where the propagation times may be different, and so an inconsistent data matching can occur on C'.

Conversely, if an inconsistent matching can occur between C and C', it means that at least two paths link C to C'. If these paths were not separated, it would mean the following:

- Either there exists a component C'' where the paths merge before C'. Then, the inconsistent matching would occur on C'' and not on C';
- Or there exists a unique path up to C'', after which the path splits in two paths to C'. Then, the inconsistent matching would occur relatively to C'' and not to C.

□

4.2. Spindle analysis

When spindles are found, we analyze how their paths influence the data used by the sink components.

4.2.1. Maximum path time. Let us consider a path $P = (C_1, C_2, \ldots, C_n)$. We note $t_{\max}(P)$ as the maximum time between the beginning of the execution of C_1, which sends a value v and the use by C_n of a value influenced by v through the path P.

To find the maximum path time, the worst case is when each component uses data at the beginning of its period and sends data at the end and when the phase difference maximizes the lag between the sending and the use of a value.

The maximum time $t'_{\max}$ between the beginning of the execution of C_1, which sends a value a to C_2, and the use by C_2 of this value through the path P depend on the time during which C_2 can use a. As shown in Figure 3,

$$t'_{\max} = 2T_{C_1} + \Delta_{C_1C_2}.$$

First, C_1 executes a step S_1 that lasts one period and produces the value a. This value can be used by C_2 until it receives a new value. A new value b is produced by C_1 at the end of the step S_2. This new value is available for C_2 after a delay $\Delta_{C_1C_2}$. We place C_2 such that it starts a step at this moment. Thus, at the same time, C_2 starts a step, and a new value is available. As we do not know what happens exactly, we choose the worst case for $t'_{\max}$, which is that C_2 starts its computation without reading the value b.

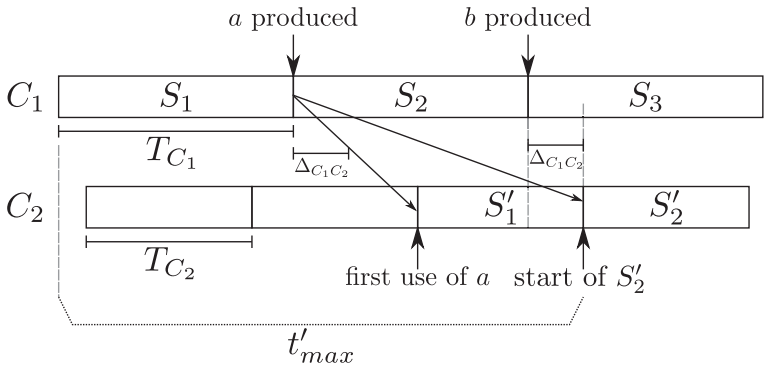

Figure 3. Evaluation of $t'_{\max}$.

The maximum time between the beginning of the execution of C_1, which sends a value v, and the use by C_n of a value influenced by v through the path P is as follows:

$$t_{\max}(P) = \sum_{i=1}^{n-1} (2T_{C_i} + \Delta_{C_i C_{i+1}}). \tag{3}$$

4.2.2. Minimum path time. Let $P = (C_1, C_2, \ldots, C_n)$ be a path. We note $t_{\min}(P)$ as the minimum time between the beginning of the execution of C_1, which sends a value v, and the use by C_n of a value influenced by v through the path P. $t_{\min}(P)$ is the sum of the minimum execution times and the minimum communication delays along the path:

$$t_{\min}(P) = \sum_{i=1}^{n-1} (e_{C_i} + \delta_{C_i C_{i+1}}). \tag{4}$$

4.2.3. Maximum gap between two input data. Let us consider a spindle between C_α and C_β composed of two paths: $P_A = (C_\alpha, C_2, \ldots C_{n-1}, C_\beta)$ and $P_B = (C_\alpha, C'_2, \ldots C'_{m-1}, C_\beta)$. P_A has a size of n, and P_B has a size of m. C_{n-1} sends a value A to C_β, and C'_{m-1} sends a value B.

C_β uses the values A and B. They are influenced by values produced by C_α. We analyze the time gap between the starting time of the step of C_α that produced the value that influences A and the starting time of the step of C_α that produced the value that influences B.

The maximum gap, noted gap_{AB}, is obtained when A is produced using the maximum path time and B is produced using the minimum path time. Moreover, A is read by C_β at the beginning of its period, and B is read as late as possible:

$$gap_{AB} = t_{\max}(P_A) + T_\beta - e_\beta - t_{\min}(P_B). \tag{5}$$

If gap_{AB} is negative, this means that data on path P_A are always propagated faster than on path P_B. Note that $gap_{AB} \neq gap_{BA}$, and both may be positive. To analyze a spindle, we have to know both values gap_{AB} and gap_{BA}.

5. DATA-MATCHING MANAGEMENT

By analyzing every spindle in the component graph, we are able to know the worst gap that we can have between two data. For the analyzed application, the designer has to decide if this gap is acceptable. If not, the objective is to reduce it.

5.1. Imposed delay

Reducing gap_{AB} can be achieved by introducing a delay into the path P_B to increase $t_{\min}(B)$. To reach this effect, queues are set on the sink component input. For every execution step, the sink uses the queue head. Data entering the queue take time to propagate to the head depending on the size of the queue. This approach is not so far from an SDF solution.

This lag increases $t_{\min}(B)$ (therefore decreasing gap_{AB}), but it also increases $t_{\max}(B)$, and so it increases gap_{BA}. We have to take care of these two effects before using an imposed delay. Moreover, we are never able to guarantee that the set used by the sink component is strictly consistent.

5.2. Timestamping

To compose a consistent set with a given consistency tolerance, the sink component must be able to select which data it needs among the received ones. Queues are used on the inputs of the sink, and for each step, the sink has the choice among the data kept in the queues. Thus, the sink needs to know the influence past of a value, which is the same as the influence past of the internal event that has produced this value.

5.2.1. Marks. A *mark* is a couple $\langle Component\_Id, value\rangle$, where values are taken from any infinite set. In practice, each component has a logical clock $H(C)$ that marks the data produced by the component. This clock 'counts' the number of computation steps executed by the component. Thereby, one mark, noted M_i, corresponds to a unique internal event i and conversely.

5.2.2. Timestamps. A *timestamp* is a set of marks that hold the influence past of an event. The timestamp carried by the event a is noted as E_a. The following timestamping rules are used:

- The set $\text{Input}(i_C)$ is composed by all the delivery events that were used to compute the internal event i_C:

$$\text{Input}(i_C) \triangleq \{d_C^{C'} : (d_C^{C'} \prec i_C \wedge \nexists D_C^{C'} : d_C^{C'} \prec D_C^{C'} \prec i_C)\}.$$

- The timestamp of a delivery event is equal to the corresponding sending event timestamp:

$$E_{d(m)} = E_{e(m)}.$$

- The timestamp of a sending event is equal to the timestamp of the most recent internal event that precedes it:

$$E_e = E_i \text{ such that } i \prec e \wedge \nexists i' : i \prec i' \prec e.$$

- The timestamp of an internal event i of a component C is equal to the union of timestamps of the delivery events used during this computation step, added to its own mark

$$E_i = \bigcup_{d \in \text{Input}(i)} E_d \ \cup \{\langle C, H(C)\rangle\}$$

  and $H(C)$ is incremented.

Lemma 1 (Marks and timestamps)

$$E_i = \{M_i\} \cup \{M_j \mid j \rightarrow^* i\}.$$

Proof
We note

$$\begin{aligned} i \overset{1}{\mapsto} i' &\triangleq \exists s, d : i \rightarrow s \rightarrow d \rightarrow i' \\ i \overset{n}{\mapsto} i' &\triangleq \exists i'' : i \overset{1}{\mapsto} i'' \wedge i'' \overset{n-1}{\mapsto} i' \quad (\text{for } n > 1), \end{aligned}$$

where i and i' are internal events. By the stamping rules, we deduce

$$\begin{aligned} E_i &= \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} E_j \\ &= \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \{M_j\} \cup \bigcup_{j \mid j \overset{2}{\mapsto} i} E_j \\ &\vdots \\ &= \{M_i\} \cup \bigcup_{n \geqslant k \geqslant 1} \bigcup_{j \mid j \overset{k}{\mapsto} i} \{M_j\} \cup \bigcup_{j \mid j \overset{n+1}{\mapsto} i} E_j \\ &\quad \text{All } \mapsto \text{ chains are bounded (initial event)} \\ &= \{M_i\} \cup \bigcup_{k \geqslant 1} \bigcup_{j \mid j \overset{k}{\mapsto} i} \{M_j\} \\ &= \{M_i\} \cup \{M_j \mid j \rightarrow^* i\}. \end{aligned}$$

□

Theorem 2
The timestamps encode the following influence relation:

$$i \rightarrow^* i' \Leftrightarrow E_i \subsetneq E_{i'}.$$

Proof
The direct implication is deduced from the stamping rules and the transitivity of $\rightarrow^*$:

$$\begin{gathered}\text{If } i \rightarrow s \rightarrow d \rightarrow i', \text{ then from the stamping rules,}\\ \Rightarrow E_{i'} = \{M_{i'}\} \cup E_i \cup X.\\ M_{i'} \text{ is unique and only comes from } i'.\\ \text{As} i' \not\rightarrow^* i \wedge i \neq i', M_{i'} \notin E_i\\ \Rightarrow E_i \subsetneq E_{i'}.\end{gathered}$$

$$\begin{gathered}\text{Using the transitivity of } \rightarrow^*,\\ i \rightarrow^* i' \Leftrightarrow i \rightarrow \ldots \rightarrow i' \Rightarrow E_i \subsetneq E_{i'}.\end{gathered}$$

The reverse implication comes from the stamping rules and Lemma 1:

$$\begin{aligned}E_i \subsetneq E_{i'} &\Rightarrow M_i \subsetneq E_{i'} \text{ (Lemma 1)}\\ &\Leftrightarrow M_i \subsetneq \{M_{i'}\} \cup \{M_j \mid j \rightarrow^* i'\} \text{ (Lemma 1).}\\ &\quad \text{A mark is unique, and } i \neq i' \Rightarrow M_i \neq M_{i'}\\ &\Leftrightarrow M_i \subsetneq \{M_j \mid j \rightarrow^* i'\}\\ &\Rightarrow \exists j : M_i = M_j \wedge j \rightarrow^* i'.\\ &\quad \text{A mark is unique:}\\ &\Leftrightarrow i \rightarrow^* i'.\end{aligned}$$

□

Theorem 3
We note $(E \mid C)$ as the subset of marks in E that were generated by the component C. An execution is consistent if and only if there do not exist several marks coming from a same component in the timestamp of each internal event:

$$\text{Strictly consistent execution} = \forall i : \forall C : \text{cardinality}(E_i \mid C) \leqslant 1.$$

Proof

$$\begin{aligned}E_i &= \{M_{i'} \mid i' \rightarrow^* i\} \cup \{M_i\}\\ &= \{M_{i'} \mid i' \rightarrow^* i \vee i' = i\}\\ &= \{M_{i'} \mid i' \in \text{past}(i)\}.\end{aligned}$$

As two distinct events cannot generate the same mark, the number of marks and the number of events are equal:

$$\begin{gathered}\text{cardinality}(\{M_{i'} \mid i' \in \text{past}(i)\} \mid C) = \text{cardinality}(\text{past}(i) \mid C)\\ \text{(with or without the restriction on } C\text{), and so}\\ \text{cardinality}(E_i \mid C) = \text{cardinality}(\text{past}(i) \mid C).\end{gathered}$$

The condition is the consistency condition defined in Equation (1). □

5.2.3. Relation to other encodings. This part of our work is in the same spirit as classical works by Lamport [18] and Mattern [19], which encode the causality relation in distributed computing. However, our influence relation is different from the usual causality relation, and we use a different encoding. The local clock does not act like a Lamport clock (the local clock is not updated using the message timestamp), and the piggybacked timestamps are not Fidge–Mattern vector clocks (we can have more than one mark from the same component).

5.2.4. Relaxed consistency encoding. The logical value in the mark identifies the step at which this mark appears. When considering relaxed consistency, we replace this logical value with the date at which the event occurs. The stamping rules are left unchanged, except that the real-time clock automatically (and possibly continuously) increases. Assuming that the precision of the clock is higher than the minimal step length, these timestamps still encode the influence pasts.

The maximal span between any two marks of a timestamp is

$$\text{span}(E) = \max_{M_i, M_{i'} \in E} (d_1 - d_2 \mid M_i = \langle \_, d_1 \rangle, M_{i'} = \langle \_, d_2 \rangle),$$

and an execution is τ-relaxed consistent if all marks coming from the same component are inside an interval of length τ:

$$\tau\text{-relaxed consistent execution} = \forall i : \forall C : \text{span}(E_i \mid C) \leqslant \tau.$$

Note that all mark comparisons, and so all date comparisons, refer to dates generated by the *same* component: The clocks of the different components do not have to be synchronized.

5.2.5. Mark generators and controllers. An inconsistent data matching can occur between a component couple (C, C') if and only if there is a spindle between them. Consequently, to reduce the number of used marks, only spindle sources are mark generators. Moreover, only sinks are controllers, which is to say components that check the consistency between the marks coming from their spindle source.

5.3. Queue handling

Data can be used as component inputs only when they make a consistent data set. It implies that data coming using the faster paths have to wait for the appropriate marked data coming through the slower paths. This requires to use queues on component inputs to store data coming faster. We analyze the necessary queue sizes in the field of relaxed data matching. As strict consistency is a relaxed consistency of tolerance 0, the results apply to strict data matching.

In the context of relaxed data matching, we use filtering queues. A filtering queue stores data it receives following a given rhythm, for example, the queue stores one value out of three. The flexibility of relaxed data matching is exploited to reduce the queue sizes by using filtering queues. A regular queue is a filtering queue with a rhythm of 1.

To manage the queues, we choose to keep a value until a more recent value is used. When a value is used, older data are erased, but the used one remains buffered. If the frequency of the receiver component is higher than the sender one, the receiver uses the same value for several steps.

In the general case, for a given spindle, the sink has an arbitrary number of inputs involved in this spindle. To find the necessary queue sizes on each input, we analyze the paths two by two. For each couple, the queue sizes of the two inputs are obtained. Then, for each input, we keep the highest queue size that was obtained from the analysis.

In the following, let us consider a spindle between C_α and C_β composed of two paths: $P_1 = (C'_1, C'_2, \ldots, C'_n)$ and $P_2 = (C_1, C_2, \ldots, C_m)$ where $C'_1 = C_1 = C_\alpha$ and $C'_n = C_m = C_\beta$. We tolerate a gap of τ between the step-starting times of C_α that produce values that influence the C_β inputs.

5.3.1. Queue requirements. First, we have to find where we need a queue.

- If $t_{\max}(P_1) > t_{\min}(P_2) + \tau$, then it means that the path P_2 can be shorter than the path P_1 and that the tolerance is not sufficient to reduce this gap. So, we need a queue between C_{m-1} and C_β.
- If $t_{\max}(P_2) > t_{\min}(P_1) + \tau$, then we need a queue between C'_{n-1} and C_β.
- If both conditions are true, then we need both previous queues. In this case, P_1, as well as P_2, can outperform the other path.

For the communication between components where the receiver is not a spindle sink, we use a buffer of size 1. Each new coming value replaces the previous one.

5.3.2. Queue size evaluation. Let us suppose that $t_{\max}(P_1) > t_{\min}(P_2) + \tau$. The required and sufficient filtering queue size between C_{m-1} and C_β must be determined. The objective is to determine the maximum number of data that must be buffered waiting for a consistent data set to be constructed. We seek this maximum size such that, when the queue is full and a new value comes, it is guaranteed that the consistent value corresponding to the oldest value v will never arrive. Thus, v is useless and can be removed.

The required queue size between C_{m-1} and C_β corresponds to the maximum number of data that can be stored in the queue between two data removals by C_β.The worst case happens when the maximum path time is made by P_1 and where P_2 takes as little time as possible.

Let us assume that C_β uses a regular queue for the path P_1 and a filtering queue for the path P_2. This filtering queue has a size of N' ($N' \geq 2$), and it stores one value out of R.

We use a simple example to present the characteristics of filtering queues. Figure 4 displays a spindle between the components C_1 and C_4 and the parameters of the components. We use simplified parameters to make easier the illustration. The input of C_4 coming from C_3 uses a filtering queue. We analyze the black path, and we consider that the communication times between two components are equal to 1 and that the filtering queue stores one value out of three.

Figure 5 illustrates the biggest gap that we can have between two step-starting times of the source that produces values that influence two consecutive values in the filtering queue. The tail of an arrow corresponds to the time when a value is produced, and its head corresponds to the time when it is read by another component or concerning the value produced by C_3, the time when the value is recorded into the queue. The biggest gap is found when the first queued value comes from a source step as old as possible and the second value comes from a step as recent as possible.

If we consider the path $P_2 = (C_1, \ldots, C_{m-1}, C_\beta)$ with a filtering queue that stores one value out of R between C_{m-1} and C_β, the biggest gap between two consecutive data is as follows:

$$\begin{aligned} gap_{filter}(P_2) &= \sum_{i=1}^{m-2} [2T_{C_i} + \Delta_{C_i C_{i+1}}] + (R+1)T_{C_{m-1}} - e_{C_{m-1}} - \sum_{i=1}^{m-2} [e_{C_i} + \delta_{C_i C_{i+1}}] \\ &= t_{\max}(P_2) - t_{\min}(P_2) + (R-1)T_{C_{m-1}} - (\Delta_{C_{m-1}C_\beta} - \delta_{C_{m-1}C_\beta}). \end{aligned}$$

If τ is the expected tolerance, we can find the necessary recording rhythm R of the queue. The relation between τ and R is provided by the gap_{filter} computation. To manage a tolerance of τ, we need that

$$gap_{filter}(P_2) \leqslant 2\tau.$$

We deduce that rhythm R of the filtering queue must respect

$$R \leqslant \max\left(1, \frac{2\tau - t_{\max}(P_2) + t_{\min}(P_2) + \Delta_{C_{m-1}C_\beta} - \delta_{C_{m-1}C_\beta}}{T_{C_{m-1}}} + 1\right). \tag{6}$$

If the rhythm of the filtering queue does not respect this condition, it is impossible to manage a relaxed consistency of tolerance τ.

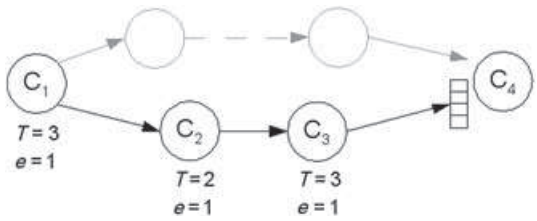

Figure 4. Example of spindle.

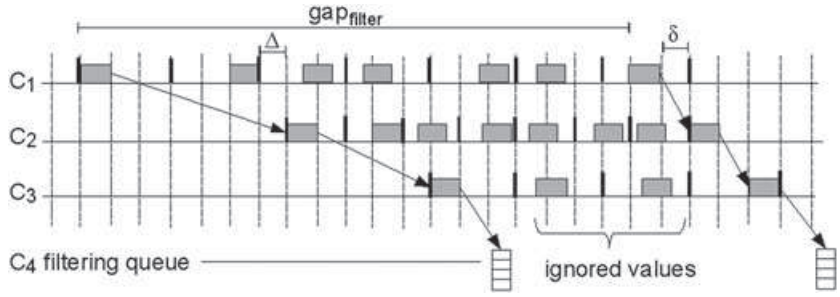

Figure 5. Behavior of a filtering queue.

Knowing the value of R, we compute the necessary queue size. When the sink reads a value in the filtering queue at time t, the oldest value it can use is influenced by a value produced by a source step that necessarily started before $t - t_{\text{oldmin}}(P_2)$:

$$\begin{aligned} t_{\text{oldmin}}(P_2) &= \sum_{i=1}^{m-2} [e_{C_i} + \delta_{C_i C_{i+1}}] + (R-1)T_{C_{m-1}} + e_{C_{m-1}} \\ &\quad + (N'-2)RT_{C_{m-1}} + e_{C_{m-1}} + \delta_{C_{m-1}C_\beta} \\ &= t_{\min}(P_2) + e_{C_{m-1}} + (N'R - R - 1)T_{C_{m-1}}. \end{aligned}$$

At time t, in the worst case, on the path P_1, the sink has a value influenced by a value produced by a source step that started at $t - t_{\max}(P_1)$.

To manage data matching, we have to provide a corresponding value coming from the path P_2. Considering relaxed data matching, this corresponding value must be influenced by a source value that was produced by a source step that started between $(t - t_{\max}(P_1) - \tau)$ and $(t - t_{\max}(P_1) + \tau)$.

The minimum queue size is obtained when

$$t_{\text{oldmin}} = t_{\max}(P_1) - \tau.$$

The necessary queue size of the filtering queue is

$$N' = \left\lceil \frac{t_{\max}(P_1) - \tau - t_{\min}(P_2) - e_{C_{m-1}} + (R+1)T_{C_{m-1}}}{RT_{C_{m-1}}} \right\rceil.$$

This queue size allows to store enough data between C_{m-1} and C_β to guarantee a data matching with values coming from the path P_1. But, we have also to take into account the worst data utilization case of the sink. The worst case happens when the data are used at the beginning of a C_β period and when the use of a new data is performed as late as possible, which is to say when the component has only its execution time left. So, in the worst case, a data is erased $2T_{C_\beta} - e_{C_\beta}$ after its last use.

If $2T_{C_\beta} - e_{C_\beta} > RT_{C_{m-1}}$, we have to add space to store the data that can come into the queue between two sink readings. The final necessary queue size N is

$$N = \left\lceil \frac{t_{\max}(P_1) - \tau - t_{\min}(P_2) - e_{C_{m-1}} + (R+1)T_{C_{m-1}}}{RT_{C_{m-1}}} + \frac{2T_{C_\beta} - e_{C_\beta}}{RT_{C_{m-1}}} \right\rceil. \tag{7}$$

With strict consistency and a regular queue, $R = 1$, and $\tau = 0$. The necessary queue size becomes

$$N = \left\lceil \frac{t_{\max}(P_A) - t_{\min}(P_B) + 2T_{C_\beta} - e_{C_\beta} - e_{C_{m-1}}}{T_{C_{m-1}}} \right\rceil + 2.$$

5.4. Application example analysis

We apply the previous results on the application example of Figure 2. For the spindles where the path temporal parameters are not very different, the necessary queues to manage strict consistency

have an average size of 6. But, the necessary queue size can be very large depending on the system parameters. If we want a strict consistency in the spindle between the position computation and the alert management, we found that we need a size of 102 on the alert management input that is directly linked with the position computation. This happens because we have a large difference between the period of the position computation (60 ms) and the one of alert management (1 s). Actually, we do not need to send the exact satellite position with the alert sent to the ground. We place a filtering queue between the position computation and the alert management. If we tolerate a gap τ of 300 ms, a recording rhythm of 9 is sufficient. The necessary queue size is then 12, far below 102, which is the size that was required for the same spindle with strict data matching.

In some cases, freshness has priority, as in the spindle between the environment and the attitude computation. To compute the attitude as precisely as possible, the component has to use the most recent data coming from the gyroscope and the star tracker. Selecting data considering the matching on the environment has no sense here.

If the queue sizes are unacceptable with regard to the resource constraints, the architecture has to be modified. Very large queue sizes are a hint that points to an architectural problem. For example, if we want a strict consistency between the position computation and the alert management, this leads to a very large queue. Instead of having a direct link between this two components, the data sent by the hot point coordinate computation can be composed of the coordinates and the position value. Thus, we can erase the link between the position computation and the alert management and eliminate the spindle.

6. CONCLUSION

In this article, we identify an important aspect of component-based distributed systems that is not treated in other works: matching of interdependent data. Our analysis is carried out as soon as the components, their characteristics, and their relations are known, but we consider few constraints on the system scheduling, so this allows us to analyze systems early in their development process.

We first detect the configurations that cause data-matching problems, and then we propose a method to manage data matching by using a timestamping mechanism to identify dependencies between data. We propose a notion of relaxed data matching and compute the necessary sizes of the queues we have to use on component inputs to manage these constraints. As strict data matching is a special case of relaxed data matching, the results are applicable to strict data matching.

In some systems, the computed queue sizes are too large with regards to the resource constraints. If this situation happens, it means that the paths are too much unbalanced. It identifies that an architecture redesign is needed. On the other hand, if the queue sizes are acceptable, it means that data matching is guaranteed independently from the final system scheduling. An open question is how more precise information about the scheduler can be used to reduce the queue sizes, for instance, by asserting that certain inconvenient executions are actually prevented from happening. Another question is whether a less regular recording rhythm such as (m, k)-firm [20] may be more efficient and more suitable to model real-time network communication.

REFERENCES

1. Möller A, Åkerholm M, Fredriksson J, Nolin M. Evaluation of component technologies with respect to industrial requirements. *30th EUROMICRO Conference,* IEEE Computer Society, 2004; 56–63.
2. Szyperski C. *Component Software – Beyond Object-oriented Programming*, 2nd ed. Addison-Wesley: New York, NY, USA, 2002.
3. Liu JWS. *Real-time Systems*. Prentice Hall: Upper Saddle River, NJ, USA, 2000.
4. Bhattacharyya SS, Murthy PK, Lee EA. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems* 1999; **21**:151–166.
5. Fong C. Discrete-time dataflow models for visual simulation in Ptolemy II. *Master's Thesis*, Electronics Research Laboratory, University of California, Berkeley, 2001.
6. Ramamritham K, Son SH, DiPippo LC. Real-time databases and data services. *Real-Time Systems* 2004; **28**(2-3):179–215.
7. Xiong M, Han S, Lam K. A deferrable scheduling algorithm for real-time transactions maintaining data freshness. *26th IEEE Real-time Systems Symposium (RTSS 2005)*, 2005; 27–37.

8. Jha AK, Xiong M, Ramamritham K. Mutual consistency in real-time databases. *27th IEEE Real-time Systems Symposium (RTSS 2006)*, 2006; 335–343.
9. Gustafsson T, Hansson J. Data freshness and overload handling in embedded systems. *12th IEEE Conference on Embedded and Real-time Computing Systems and Applications (RTCSA 2006)*, 2006; 173–182.
10. Xiong M, Sivasankaran R, Stankovic J, Ramamritham K, Towsley D. Scheduling transactions with temporal constraints: exploiting data semantics. *17th IEEE Real-time Systems Symposium (RTSS'96)*, 1996; 240–253.
11. Anderson S, Filipe JK. Guaranteeing temporal validity with a real-time logic of knowledge. *23rd Conference on Distributed Computing Systems (ICDCS 2003),* IEEE Computer Society, 2003; 178–183.
12. Song XC, Liu JWS. Maintaining temporal consistency: pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering* 1995; **7**(5):786–796.
13. Urgaonkar B, Ninan AG, Raunak MS, Shenoy PJ, Ramamritham K. Maintaining mutual consistency for cached web objects. *21st International Conference on Distributed Computing Systems (ICDCS-21),* IEEE Computer Society, 2001; 371–380.
14. Baldoni R, Prakash R, Raynal M, Singhal M. Efficient Δ-causal broadcasting. *International Journal of Computer Systems Science and Engineering* 1998; **13**(5):263–269.
15. Pontisso N, Padiou G, Quéinnec P. Real time data consistency in component based embedded systems. *8th International Conference on New Technologies in Distributed Systems (NOTERE '08),* ACM, 2008; 1–6.
16. Pontisso N, Quéinnec P, Padiou G. Temporal data matching in component based real time systems. *IEEE Symposium on Industrial Embedded Systems SIES2009*, 2009; 62–65.
17. Pontisso N, Quéinnec P, Padiou G. Analysis of distributed multi-periodic systems to achieve consistent data matching. *10th Annual International Conference on New Technologies of Distributed Systems NOTERE 2010*, 2010; 81–88.
18. Lamport L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
19. Mattern F. Virtual time and global state in distributed systems. *International Workshop on Parallel and Distributed Algorithms,* Elsevier, 1989; 215–226.
20. Jia N, Song YQ, Lin RZ. Analysis of networked control system with packet drops governed by (m,k)-firm constraint. *6th IFAC International Conference on Fieldbus Systems and Their Applications (FeT'2005),* Elsevier, 2005; 63–70.